Supporting Experimental Computer Science

Frédéric Desprez, Geoffrey Fox, Emmanuel Jeannot, Kate Keahey, Michael Kozuch, David Margery, Pierre Neyron, Lucas Nussbaum, Christian Pérez, Olivier Richard, et al.

Supporting Experimental Computer Science

Editors: Kate Keahey, Frédéric Desprez

Authors: Frédéric Desprez, Geoffrey Fox, Emmanuel Jeannot, Kate Keahey, Michael Kozuch, David Margery, Pierre Neyron, Lucas Nussbaum, Christian Perez, Olivier Richard, Warren Smith, Gregor von Laszewski, Jens Vöckler

Published as ANL MCS Technical Memo 326

Table of Contents

1. Introduction

The ability to conduct consistent, controlled, and repeatable large-scale experiments in all areas of computer science related to parallel, large-scale, or distributed computing and networking is critical to the future and development of computer science. Yet conducting such experiments is still too often a challenge for researchers, students, and practitioners because of the unavailability of dedicated resources, inability to create controlled experimental conditions, and variability in software. Availability, repeatability, and open sharing of electronic products are all still difficult to achieve.

To discuss those challenges and share experiences in their solution, the Workshop on Experimental Support for Computer Science [1] brought together scientists involved in building and operating infrastructures dedicated to supporting computer science experiments to discuss challenges and solutions in this space. The workshop was held in November 2011 and was collocated with the SC11 conference in Seattle, Wash. Our objectives were to share experiences and knowledge related to supporting large-scale experiments conducted on experimental infrastructures, understand user requirements, and discuss methodologies and opportunities created by emerging technologies.

This report ties together the workshop presentations and discussion and the consensus that emerged on the state of the field and directions for moving forward. In Section 2 we set the stage by describing the experimental culture and existing methodology in computer science. In Section 3, we describe the properties of the experimental testbeds whose representatives were participating in the workshop—Grid'5000 in France and Future Grid and Open Cirrus in the United States—as well as the projects that these testbeds support. The layers of experimental infrastructure are described in Section 4, followed in Sections 5 and 6 by profiles of tools and approaches taken by the respective testbeds to provide basic experiment management services and experiment orchestration. In Section 7 we summarize the workshop findings.

2. Experimental Methodology for Computer Science

Compared with physics, biology, or mathematics, the discipline of computing is one of the last being introduced in higher education curriculum. Because of this relative youth, the status of computer science gives rise to much discussion among practitioners and outsiders. Indeed, compared with a physicist, a biologist or a mathematician, a *computer scientist* is not always seen as a scientist but rather as an engineer or a programmer.

Historically, computer science is an offspring of mathematics, but it differs from this discipline by its direct applications and object of study (computers, algorithms, the Internet, etc.) and by its interdisciplinary aspects: it has many links with the natural sciences (e.g., biology, geology, and energy sciences) but also with the social sciences (e.g., sociology, linguistics, and law). In [2] the former ACM chair, Peter J. Denning, examined this new autonomous discipline and concluded that computer science meets all the criteria of a scientific discipline. The goal of this discipline is to gather and organize a set of knowledge [3]:

"The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis design, efficiency, implementation and application."

While it is important to understand that computer science is also partly engineering and technology, it is striking to see that for many aspects of this discipline experiments play a key role.

2.1. **Importance of Experiments in Computer Science**

In "What Is Experimental Computer Science?" [4] Denning studies the role of experiment of computer science and shows that in some ways computer science is comparable to other sciences in terms of the role and the importance of experiment.

2.1.1. *Necessity of Experiment in Computer Science*

As an offspring of mathematics, originally computer science was considered a formal science: definitions are stated and theorems demonstrated. However, experiments also are necessary in computer science:

1. A formal science is based on *models*. Computer science, like in many other sciences, involves the use of models. Since models are a description of reality, it is important to assess these models and test their validity. In order to assess a model, the scientific methodology consists of making hypotheses and testing them through experiments. If the experiments invalidate the hypothesis, the model is flawed. A good example of such model invalidation is the work of Paxson et al. [5]. They showed that the well-used Poisson model of the wide-area TCP arrival process was not matched in many cases and that other models (long-tail distributions) must be used.
2. Computer science studies many objects in order to understand them better. This is the case with hardware (CPU, disks, etc.), programs, data, protocols, algorithms, networks, and so forth. As the technology develops, these objects become increasingly complex. For instance, a distributed-computing infrastructure is composed of several layers (hardware, runtime system, programming environments, applications, etc.) built on top of each other. Understanding such a system requires careful modeling of each layer and the interactions between them. Since the complexity of each layer is already extremely high, it is not feasible to build a precise model of the whole environment. In this case, experiments are necessary to isolate parts of the holistic behavior in order to understand a specific portion of the whole.

In [6], Tichy identified two advantages of experimentation. First, by testing hypotheses, algorithms, or programs, experiments help to construct a database of knowledge on theories, methods, and tools used for such study. Second, some observations lead to unexpected or negative results, which help eliminate some less fruitful fields of study, erroneous approaches, or false hypotheses.

2.1.2. *Experimental Culture in Computer Science*

Being a young science, computer science has not yet developed the culture of experimentation present in other sciences. For example, Luckowicz et al. [7] studied papers published in the ACM journals in the 1990s and concluded that between 40% and 50% of the papers requiring experimental validation had none. Four years later Zelkowitz and Wallace [8] reported on a study of 622 papers published between 1985 and 1995 and concluded that even though the situation was improving, "too many articles [still had] no experimental validation."

Today, experimental validation is given more weight, but the quality is not always at the level of other sciences. For instance, we studied the papers with graphs having error bars in the Euro-Par conference series, one of the leading conferences in the domain of parallel computing. The results, listed in Table 1, show that in the studied period the number of papers with graphs containing error bars is around 5%. We are not claiming that all the papers should have graphs with error bars: some papers present only theoretical results, and some papers study error in their experiments by other means. But we estimate that around 70% of the papers should have had error bars. This situation is not necessarily specific to the Euro-Par conference series (we selected this conference series for our study because it provides printed proceedings that are easier to check); similar issues are likely to be present in material submitted to other serious conferences.

However, this is not the case in many other scientific domains, such as in physics, where error analysis is a recognized part of experimentation. Therefore, we see that computer science does not have the same standards as other sciences in terms of the quality of experimentation.

Table 1: Papers published with error bars at the Euro-Par conference (2007–2011)

Euro-Par	No. of Papers	With Error Bars	Percentage
2007	89	5	5.6
2008	89	3	3.4
2009	86	2	2.4
2010	90	6	6.7
2011	81	7	8.6
2007-11	435	23	5.3

2.1.3. *Properties of a Good Experiment*

To improve the situation, computer science practitioners must pay closer attention to the quality of their experiments and its documentation. In [9], the authors described the properties that a good experiment should have:

- *Reproducibility.* This is the basis of the experimental methodology. An experiment must give the same result with the same input.
- *Extensibility.* An experiment must target possible comparisons with other works and extensions (the use of more or different processors, larger data sets, or different architectures).
- *Applicability.* An experiment must define realistic parameters and must allow for an easy calibration.
- *Revisability*. When an implementation does not perform as expected, an experiment must help identify the reasons.

In [10] chapter 2, Jain presented a *systematic approach to performance evaluation* that explains how to design an experiment, from the definition of system boundaries to the presentations of results and the workload or metric selection.

2.1.4. *Experiment Workflow*

A good experiment will have the following workflow:

- *Hypothesis.* Creating a hypothesis is the foundation of any scientific experiment. The hypothesis expresses what the scientist thinks is true.
- *Apparatus.* The scientist then designs an experiment to either prove or disprove the hypothesis. An important part of the design is to document the experimental setup, which we call here the *apparatus* [4].
- *Execution.* Part of the experiment is the placement and density of sensors to record sufficient data to substantiate (or disprove) the claim from the hypothesis, as well as document all observations.
- *Conclusion.* With the record of the sensors, a conclusion can be reached. Often, this conclusion is presented in the form of a paper or other publication. The documentation at

this point should be sufficient to enable other scientists to repeat the experiment in their own labs.

2.2. **Taxonomy of Experimental Methodologies**

We have seen that careful design of experiments is important in order to obtain meaningful results. Equally important, however, is having robust and easy-to-use tools and environments for performing these experiments.

Designing experiments can sometimes be cumbersome because of the number of parameters to manage and the environment in which the experiment is conducted. Inn order to ease the design and the execution of experiments, several methodologies have been proposed. The authors of [11] have proposed a unified taxonomy to classify experimental methodologies. It is based on two components: the application that is tested and the environment on which it is tested. Either one or both of these two components could be represented by a model or could be real. Thus, we can identify four classes of experiment.

- In *in situ* experiments, a real application is evaluated on a real platform. Such a real environment can be of different scale, but the application is run unmodified on it in order to test it under real conditions. For instance, in Grid'5000 [12], real programs are executed on a distributed environment to evaluate their resilience or scalability.
- In *emulation* experiments, a real application is executed on a model of a platform. For instance, in Wrekavoc [13], by slowing network links, a wide-area network is emulated on cluster.
- In *benchmarking* experiments, a model of an application is executed on a real platform. For example, the NAS benchmarks [14] are used to evaluate the performance of a parallel machine, but contrary to in situ experiments the results of the computation are not relevant.
- In *simulation* experiments, a model of an application is executed on a model of an environment. For instance, Simgrid [15] is used to simulate distributed applications on a parallel heterogeneous system.

These types of experiments (together with software that facilitates them) are depicted in Figure 1.

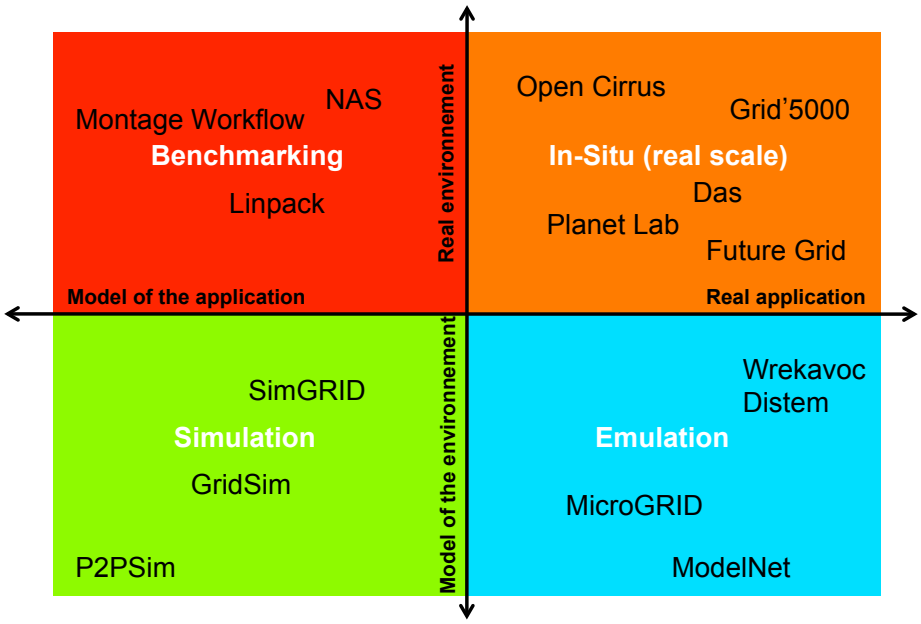

Figure 1: Four methodologies for computer science experiments

Choosing the correct environment for conducting an experiment is crucial for the quality of the results and their interpretation. If one wants to evaluate hardware, benchmarking is well suited; but if one wants to evaluate an application on a system that is not yet available, emulation or simulation is necessary.

3. Experimental Testbeds

Several testbeds have been established to support experimental computer science. In this section we profile three of them: Grid'5000, FutureGrid, and Open Cirrus. We describe their respective goals, resources offered, and support for specific experiment types, and we discuss lessons learned from their use.

3.1. **Grid'5000**

Grid'5000 [16, 17] is located mainly in France, with one operational site in Luxembourg and a second site, not implementing the complete stack, in Porto Alegre, Brazil. Grid'5000 provides a testbed supporting experiments on various types of distributed systems (high-performance computing, grids, peer-to-peer systems, cloud computing, and others), on all layers of the software stack shown in Figure 2. The project was started in 2003 (funded by an initiative from French ministry of research), and the testbed has been opened to users since 2005.

The core testbed currently comprises 10 sites, as shown in Figure 3. Grid'5000 is composed of 26 clusters, 1,700 nodes, and 7,400 CPU cores, with various generations of technology. A dedicated 10 Gbps backbone network is provided by RENATER (the French National Research and Education Network). In order to prevent Grid'5000 machines from being the source of a distributed denial of service, connections from Grid'5000 to the Internet are strictly limited to a list of whitelisted data and software sources, updated on demand.

The funding for hardware and engineering comes from the French research organization INRIA (through the Aladdin-G5K project), CNRS *(*Centre national de la recherche scientifique), and many universities throughout France. The funding structure differs at each site, but in the most common case INRIA provides funding for hardware, operated by a combination of partner universities, CNRS, and INRIA. Human resources and travel expenses are largely covered by INRIA.

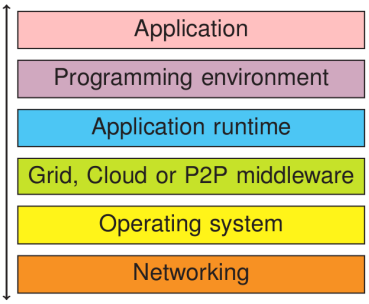

Figure 2: Grid'5000: supporting experiments on all layers of the software

Figure 3:Grid'5000 map

From the user point of view, Grid'5000 is a homogeneous set of sites, with the exact same software environment. The driving idea is that users willing to face software heterogeneity should add controlled heterogeneity themselves during their experiments.

Three basic workflows are supported when staging an experiment on Grid'5000: a web interface-based workflow, an API-based workflow, and a shell-based workflow. These differ not only in the interfaces used but also in the process they support.

The core steps identified to run an experiment are (1) finding and booking suitable resources for the experiment and (2) deploying the experiment apparatus on the resources. *Finding suitable resources* can be approached in two ways: either users browse a description of the available resources and then make a booking, or they describe their needs to the system that will locate appropriate resources. We believe both approaches should be supported, and therefore a machine-readable description of Grid'5000 is available through the reference API. It can be browsed by using a web interface or by running a program over the API. At the same time, the resource scheduler on each site is fed with the resource properties so that a user can ask for resources describing the required properties (e.g., 25 nodes connected to the same switch with at least 8 cores and 32 GB of memory). Once matching resources are found, they can be reserved either for exclusive access at a given time or for exclusive access when they become available. In the latter case, a script is given at reservation time, as in classical batch scheduling.

Different approaches to *deploying the experimental apparatus* are also supported. At the infrastructure level users either use the preconfigured environment on nodes, called the production environment, or they install their own environment. An environment consists of a disk image to be copied on the node and of the path in the disk image of the kernel to boot. This environment can be prepared in advance by modifying and saving reference environments made available to users, or a reference environment can be dynamically customized after it is deployed

on the resources. The approach chosen can affect the repeatability of the results. Therefore, choices concerning the experiment testbed environment are left to the experimenters.

Whatever approach used for the first two steps described here, access to resources (sites and nodes) is done through SSH. Each site has its own NFS server, as shown in Figure 4.This design decision was taken to ensure that resources of a particular site can be used even when the link to other sites is undergoing maintenance. In other words, the infrastructure does not depend on a single site to stay operational—an important consideration because maintenance events become frequent when 10 sites are operated.

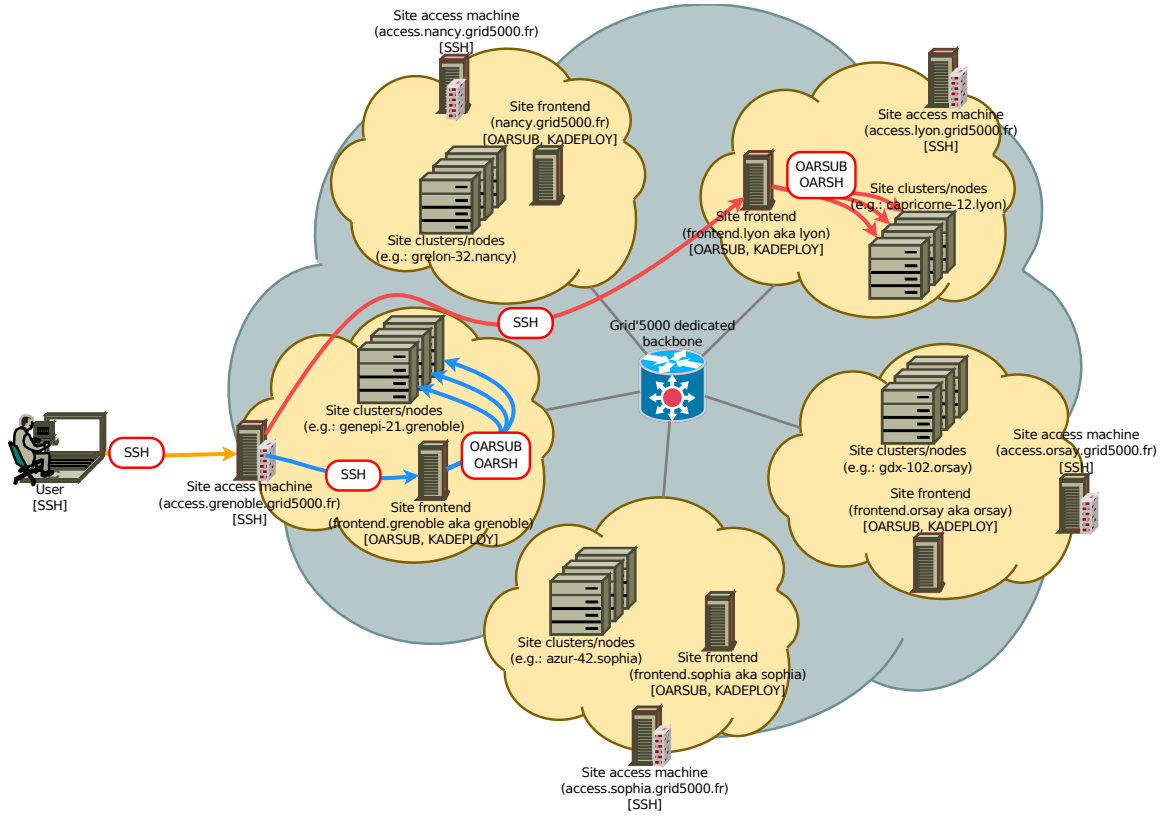

Figure 4: User access to Grid'5000 facilities

Several tools are provided to facilitate experiments. Most of them were originally developed specifically for Grid'5000.

- Grid'5000 users select and reserve resources with the OAR batch scheduler [18].
- Users can install their own system image on the nodes (without any virtualization layer) using Kadeploy [19].
- Experiments requiring network isolation can use KaVLAN to reconfigure switches and isolate nodes from the test of the testbed (Figure 5).
- Several monitoring tools (resource usage on nodes with Ganglia, energy consumption) are also available.

All tools can be accessed by a REST API to ease the automation of experiments using scripts.

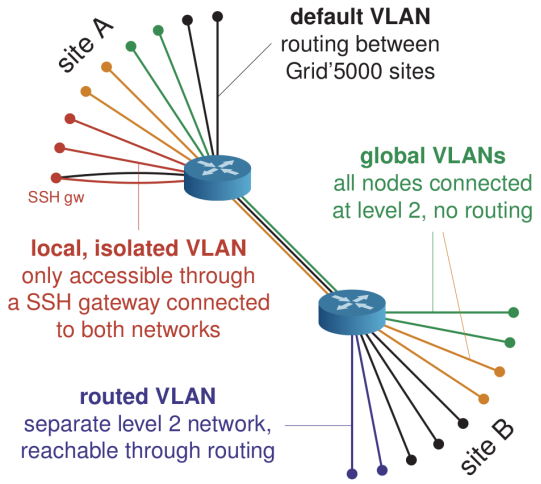

Figure 5: KaVLAN: level-2 network isolation tool

More than 578 publications relied on Grid'5000 for their experiments. For example, in 2009, Grid'5000 contributed to the factorization of RSA-768 [20], breaking a new record in integer factorization. Grid'5000 also played a key role in understanding the performance features of the algorithms thanks to the variety of available hardware technologies. In 2011, a Nimbus cloud was deployed on hundred of nodes of Grid5000 and was connected to FutureGrid for a large-scale demonstration of "sky computing"—computing on a testbed of federated clouds [21].

Despite the availability of the Grid'5000 experimental testbed, however, no structured community existed to exchange experiences around such a tool. Therefore, the INRIA large-scale initiative Hemera [22] was established to gather researchers in order to address scientific challenges involving ambitious scaling of techniques for large-scale distributed computing, to revitalize the scientific community around Grid'5000, and to enlarge the Grid'5000 community by helping newcomers use the testbed.

To this end, Hemera formulates scientific challenges that involve carrying out several multi-dimensional experiments on the Grid'5000 testbed. The current open challenges cover different research fields; the four largest ones (1) profiling of energy consumption of large-scale applications, (2) testing of production grid software (in particular, grid software such as gLite [69]), (3) combinatorial optimization problems, and (4) experiments related to multiparametric-intensive stochastic simulations for hydrogeology.

In addition, Hemera has organized several working groups to allow intellectual exchange on long-term issues. At the time of this writing, four such groups have been established related to classical large-scale computer science: (1) transparent, safe, and efficient large-scale computing, (2) network metrology and traffic characterization, (3) efficient management of large volumes of information for data-intensive applications, and (4) efficient exploitation of highly heterogeneous and hierarchical large-scale systems. Two other working groups focus on more recently emerged topics: (5) virtualization technologies and (6) energy efficiency. And two working groups are more strongly connected to the experimental approach: (7) one focusing on modeling large-scale systems and validating their simulators and (8) one focusing on understanding how to complete challenging experiments on Grid'5000. The last working group in particular explores different complementary approaches that are the basic building blocks for constructing the next level of experimentation on large-scale experimental platforms: the methodology of designing complex experiments, the expression of the numerous steps that compose experiments efficiently, the

configuration of the experimental platform, and the extraction of large-scale experimental results with as little intrusiveness as possible.

3.2. **FutureGrid**

The FutureGrid [23] project mission is to enable experimental work that advances four areas.

- Innovation and scientific understanding of distributed computing and parallel computing paradigms
- Engineering science of middleware that enables these paradigms
- Use and drivers of these paradigms by important applications
- Education of a new generation of students and workforce on the use of these paradigms and their applications

The implementation of this mission includes providing distributed flexible hardware with supported use, infrastructure-as-a-service (IaaS) and platform-as-a-service (PaaS) "core" software with supported use, and a growing list of software from FutureGrid partners and users. In this way the FutureGrid project enables high-performance computing systems, grids, and clouds. Topics range from programming models, scheduling, virtualization, middleware, storage systems, interface design, and cybersecurity to the optimization of grid-enabled and cloud-enabled computational schemes in astronomy, chemistry, biology, engineering, atmospheric science, and epidemiology. Since FutureGrid supports interactive use, it is well suited for testing and supporting distributed-system and scientific computing classes. Education and broader outreach activities include the dissemination of curricular materials on the use of FutureGrid, prepackaged FutureGrid virtual machines (appliances) configured for particular course modules, and educational modules based on virtual appliance networks and social networking technologies.

FutureGrid is a national-scale grid and cloud testbed facility that includes a number of computational resources at distributed locations and forms a part of NSF's national high-performance cyberinfrastructure XSEDE [24]. Partners in the FutureGrid project include Indiana University, the University of Chicago, University of Florida, San Diego Supercomputer Center, University of Southern California, University of Texas at Austin, University of Tennessee at Knoxville, and University of Virginia. Most partners contribute both hardware and software, as well as support. Tables 2 and 3 list computational and storage resources, respectively. All network links in FutureGrid are dedicated (10 GbE lines for all but to Florida, which is 1 GbE), except the link to the Texas Advanced Computing Center (TACC), part of the University of Texas at Austin. The network is unique in that it can be dedicated to conduct experiments in isolation, using a network impairment device (Spirent H10 XGEM Network Impairment emulator co-located with the core router) for introducing a variety of predetermined network conditions (see Figure 6). The significant number of distinct systems in FutureGrid provides a heterogeneous distributed architecture connected by high-bandwidth network links supporting distributed system research.

Table 2: Current compute resources of FutureGrid as of January 2012

Name	System Type	No. Nodes	No. CPUS	No. Cores	TFLOPS	Total RAM (GB)	Site
india	IBM iDataplex	128	256	1024	11	3072	IU
hotel	IBM iDataplex	84	168	672	7	2016	UC
sierra	IBM iDataplex	84	168	672	7	2688	SDSC
foxtrot	IBM iDataplex	32	64	256	3	768	UF
alamo	Dell PowerEdge	96	192	768	8	1152	TACC
xray	Cray XT5m	1	168	672	6	1344	IU
bravo	HP Proliant	16	31	128	1.7	3072	IU
Total		***441***	***1048***	***4192***	***43.7***	***14112***	

Table 3: Storage resources of FutureGrid as of January 2012

System Type	Capacity (TB)	File System	Site
Xanadu 360	**60**	**NFS**	**IU**
DDN 6620	**120**	**GPFS**	**UC**
Sunfire x4170	**96**	**ZFS**	**SDSC**
Dell MD3000	**30**	**NFS**	**TACC**
IBM dx360 M3	**24**	**NFS**	**UF**

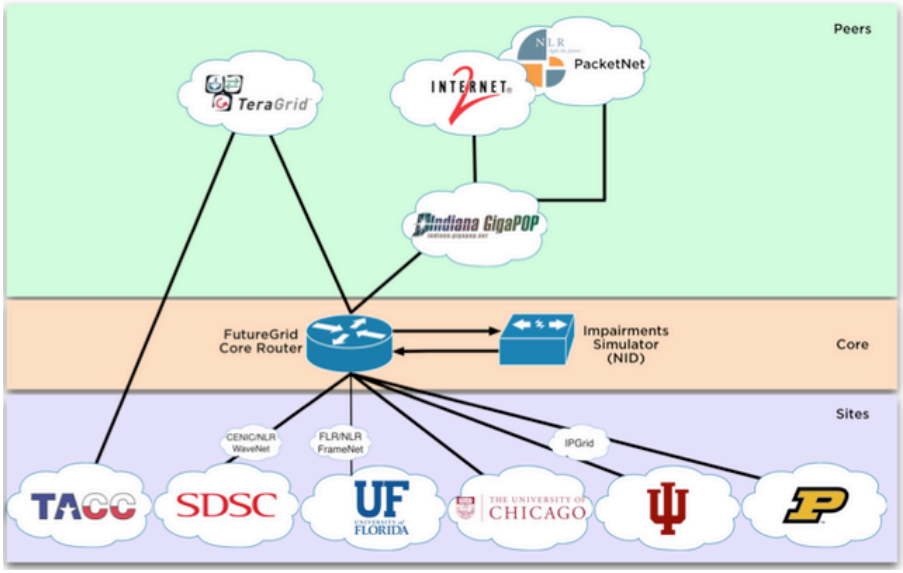

Figure 6: Network infrastructure of FutureGrid as of January 2012

Users obtain access to FutureGrid by submitting project applications describing the nature and merit of their activity. Currently most projects tax FutureGrid not in number of resources requested but in the nature and support of requested software, and these issues are taken into consideration when granting access; for example, FutureGrid does not accept projects in production science. Once a project is approved, users are given access and can conduct experiments. Currently no restrictions are placed on nationality or type (academic, government, industry) of users; however, the use of FutureGrid must be documented and the results shared with the community.

FutureGrid allows both federated and nonfederated experiments. The federated infrastructure is based on LDAP [25] where possible, while using public keys. Users are accepted based on a simple verification process relying on a search on academic publications, participation in source code development of established projects, or lookup on university Web sites.

FutureGrid is developing a number of tools that together provide a sophisticated experiment management environment. The architecture also reuses and integrates various tools, including Nimbus [26], OpenStack [27], OpenNebula [28], Eucalyptus [29], Globus [30], Unicore [31], Genesis II, and Pegasus [32]. The intention is that most systems in FutureGrid eventually will be available via dynamic provisioning, that is, reconfigured as needed by software developed as part of the FutureGrid stack, with proper access control by users and administrators. For dynamic provisioning on bare metal and VMs, FutureGrid currently uses the Rain software, developed as part of the FutureGrid project, which not only places the operating system on the resources (virtualized and nonvirtualized) but also assembles the operating system and software stack as part of an image generation process [33, 34].

Despite the relative youth of the project (currently two years in operation), as of this writing more than 170 projects are registered in FutureGrid (for a full list of projects see [35]). These projects cover a wide range, from applications to technology and from research to education. Recent projects have focused on integration testing for XSEDE, image management and dynamic provisioning on bare metal, and scalability tests for cloud provisioning. These projects are ground breaking in that they introduce a testbed environment for XSEDE and allow user facing dynamic provisioning, something not normally offered by other resources. The scalability experiments showed certain limitations with standard cloud setups for use cases typical for scientific applications.

The FutureGrid projects can be categorized as follows: 47% computer science, 27% technology evaluation, 18% life sciences applications, 13% other applications, 8% education, and a small but important 3% interoperability projects (some projects covered multiple categories; hence the total > 100%). Education is actually more important and successful than the fraction indicates because a single class project implies 20–50 users of FutureGrid.

We found that the diverse needs of users require significant user support but that many users did not need huge numbers of nodes. Thus we changed plans and targeted more funds at user support and less on hardware expansion. We also found that the ability to request both bare metal and virtualized nodes was important in many projects; this was perhaps not unexpected, but it is different from traditional environments with fixed software stacks. Further, we note that cloud technologies are rapidly changing every 3–6 months, requiring substantial effort from both software and systems groups to track, deploy, and support. These groups must collaborate closely; for example, automating processes documented by the systems team through software development is helpful in providing a scalable service.

3.3. **Open Cirrus Research Testbed**

In the summer of 2008, researchers from HP, Intel, and Yahoo! became concerned that a lack of infrastructure resources could inhibit development of vibrant academic research in system

software for cloud computing. As a result, these companies cosponsored the formation of the Open Cirrus research testbed [36]. The key observation was that, although commercial cloud offerings were available to academic researchers, building new systems in these environments is difficult because they do not provide access to the underlying hardware. Consequently, the sponsors designed the Open Cirrus effort with the goal of putting hardware resources of interesting size in the hands of academic researchers so that they can explore future cloud technologies. To enable the effort to scale, the Open Cirrus testbed was formed around a federated model; multiple institutions would each manage a computing cluster of at least 1,000 cores that would be made available to external researchers. At the time of this writing, over a dozen sites around the world participate in the Open Cirrus testbed [37].

Because the sites are individually managed, each site has a hardware and software environment tailored for the local research community. The Intel site [38], for example, provides services primarily to the research community at Carnegie Mellon University. The equipment of this site consists of approximately 200 dual-socket, rack-mounted servers spanning approximately 8 different configurations. These servers are connected to top-of-rack (TOR) switches through 1 Gb Ethernet connections, and the TOR switches are connected together through a core 10 Gb Ethernet network.

Users are given access to individual Open Cirrus sites[1] by contacting the particular site's administrative contacts, but users may execute experiments across multiple sites by acquiring accounts at the appropriate set of sites. Requesting an account is relatively straightforward; typically, a prospective user is simply asked to describe the proposed research project, the resources requested, and the expected outcomes of the research project. Preference may be given to research proposals expecting to publish their results in scientific venues and/or contribute software developed as open source artifacts. Sites often provide access, once an account is granted, through SSH (secure shell).

The Open Cirrus community recommends that each site provide a set of software services to support cloud computing research [39]; these services include a physical resource allocation service, a virtual machine resource allocation service, a distributed storage service, and one or more distributed computing frameworks. At the Intel Open Cirrus site, for example, the physical and virtual layers are provided by Zoni and Tashi, respectively (both are part of the same Apache Software Foundation effort [40]); the storage role is filled by HDFS (the Hadoop file system); and the computing frameworks include Hadoop [41] and Maui-Torque [42]—all open source software components.

The physical resource allocation service enables the core capability of placing raw hardware resources in the hands of researchers. This component is responsible for five actions. First, it manages the *allocation* of resources, particularly server nodes, to research projects. Second, it *provisions* the server nodes allocated to a project with the particular system software needed. Third, the physical resource allocation service effects the *isolation* of resources, typically through the configuration of VLANs, so that different experiments do not interfere with each other. Fourth, it provides mechanisms for the remote *management* of physical resources, particularly powering-off or powering-on the server nodes. Fifth, it provides remote *debugging* facilities, crucial when new operating system images are installed.

Site administrators may designate one of these isolated physical resource domains to hosting the production services of the site, such as the virtual resource allocation layer and the computing frameworks. By providing these services, the sites simultaneously support research projects beyond those directly concerned with cloud system software and foster the development of

[1] This policy ensured that each site could develop user admission and management policies that complied with local laws.

realistic workloads and traces that, in turn, may better inform the design of the system software projects. As a result, some significant fraction of the Open Cirrus user base uses the resource as they would any other commercial cloud computing offering.

As the Open Cirrus effort matured, two somewhat surprising aspects of the testbed emerged. First, few of the research projects capitalized on the federated design of the testbed. While the initial sponsors of the testbed expected significant research interest in the use of federation, over time it has become clear that the Open Cirrus user community has not looked upon federation as a property of great research interest (e.g., as a mechanism for providing failure resilient systems); the reasons are still unclear. Second, the Open Cirrus Summit gatherings have evolved organically as an effective mechanism for building community among the member sites. Recent events have operated like workshops—complete with paper submissions, program committees, and conference presentations—and provide a forum for the exchange of research results and experiences.

The Open Cirrus testbed has proved invaluable as a collaborative infrastructure supporting research that includes both industrial and academic contributors. One of the keys to the success of the Open Cirrus effort is the set of software services chosen as the base architecture. The physical resource allocation service provides the fundamental capability of the testbed—enabling researchers to experiment with raw hardware, such as the team reconfiguring network switches to emulate an optical, circuit-switched, data center network [43]. However, the other services are equally valuable in that they provide near-commercial-quality cloud services for nondomain research and enable the collection of realistic workloads.

4. Experiment Management Software

To ensure the experiment properties described in Section 2.1.3, experimental testbeds for computer science must provide a comprehensive set of features and services to support experimentation: frequent checks of resources to detect hardware bugs, deployment of custom system images, network isolation, and monitoring (network, power consumption). A testbed also must make APIs available to interact with those services in order to facilitate the scripting of experiments.

These features, however, are not sufficient to address experimental needs in a rapidly evolving field. As technology becomes more complex, both the complexity and the scale of experiments must be increased in order to answer harder scientific questions. Moreover, the quality of experimental processes must improve in order to increase experimental credibility and reproducibility

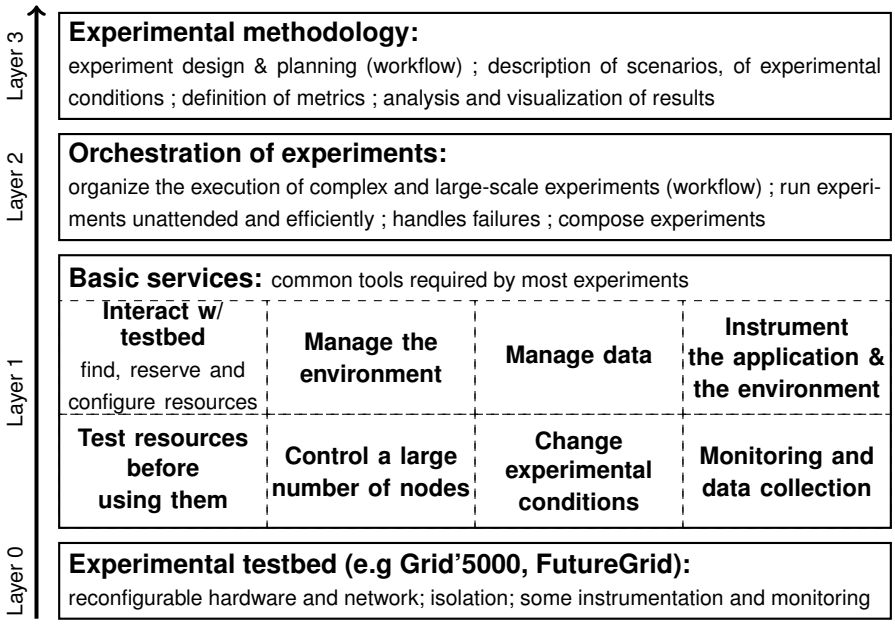

Figure 7: Layers of experiment infrastructure.

Figure 7 shows the various layers of the experimental infrastructure. Layers 0 and 3, discussed in Sections 3 and 2, respectively) are closely interrelated: while we must pursue and perfect the development of experimental testbeds, the community needs to adapt work on experimental methodology and design of experiments (e.g., [10]) to what is technically and economically feasible to realize. This feasibility is typically determined by services developed in layers 1 and 2. We identified a set of basic (layer 1) services that address the needs shared by most experiments.

- Interaction with the testbed, to select, reserve, and configure resources (with tools such as OAR or Kadeploy in the context of Grid'5000)
- Testing of reserved resources before they are used, in order to detect hardware and software problems (misconfiguration or malfunction) that could affect the experimental results
- Management of the experimental environment, addressing provenance issues to enable both its identical reconstruction and the analysis of its components
- Efficient control of a large number of nodes, with tools such as TakTuk [44]
- Data management: distribution and gathering of data required and generated by the experiment
- Change of experimental conditions (introducing heterogeneity, emulating a complex network topology, injecting load and faults) with tools such as Emulab[45] or Distem [46]
- Instrumentation of the application and the environment in order to generate traces during the experiment
- Monitoring and data collection, to extract synthetic information about the experiment

On top of those services, software is needed in layer 3 to orchestrate experiments, moving from failure-prone scripts involving many manual steps, to a formalism enabling the organization, combination and sharing of experiments. Several attempts to address this issue have already been made, mostly linked to experimental testbeds (e.g., Emulab [45, 47], PlanetLab & GENI [48],

Grid'5000—with NXE [49], Expo [50], Execo [51], and g5k-campaign [52]). Other sciences have adopted tools based on workflows (Kepler [53], Taverna [54], VisTrails [55]), which might be a solution in distributed systems research as well.

5. Basic Experiment Management Services

In this section, we profile several tools, developed by Grid'5000 and FutureGrid, respectively, that provide services in layers 1 and 2.

5.1. **Managing the Environment: Taktuk**

TakTuk is a versatile tool for application deployments on large and complex computing infrastructures; it optimizes the deployment of parallel remote executions of commands to a potentially large set of remote nodes. TakTuk uses an adaptive algorithm and sets up an interconnection network to transport commands and perform I/O multiplexing and demultiplexing. The TakTuk mechanics dynamically adapt to the environment (machine performance and current load, network contention) by using a reactive work-stealing algorithm that mixes local parallelization and work distribution.

TakTuk fulfills several requirements associated with the management of experiments on distributed experimental testbeds such as Grid'5000 and FutureGrid. TakTuk can play a significant role as a basic service (Layer 1 in Figure 7) on top of a bare experimental testbed infrastructure, providing an efficient and reliable building block to orchestrate complex experiments.

On the most basic level, TakTuk can be seen as an efficient version of a "forall hosts do ssh" loop, capable of sustaining the load of computing on clusters, grids, clouds, or even "skies" (cloud federations). Scalability and fault tolerance are ensured for up to a 1,000 nodes. Dealing with topology constraints, such as the requirement to use a gateway to reach certain nodes,, or to group nodes for a network link efficiency optimization, is possible. Beyond simple launch management functionality, TakTuk can be used to set up an overlay (communication layer) on top of which an application can exchange information without knowing the details of the underlying (possibly complex) infrastructure.

TakTuk also provides a powerful user interface, with a large number of possible customizations: connector (rsh, ssh) change; autopropagation of TakTuk's code, thereby removing the need to install it on each node; enforcement of topology constraints; and output formatting of various streams (stdout, stderr, status, and potentially others). TakTuk can be run interactively, enabling users to get the current state of the deployment tree and perform operations such as running commands, transmitting files, or deploying new nodes on the already deployed interconnection network. Moreover, those features can also easily be exploited by high-level programs through a Perl-based, event-driven API [44].

TakTuk is heavily used underneath by the Grid'5000 resource manager middleware, OAR, and the deployment middleware, Kadeploy. OAR uses TakTuk for first checking the health of the resources and then, at a job (experiment container) launch, for configuring the operating system of the nodes (e.g., cpusets, access control lists). Moreover, it is used to clean up the nodes (processes, IPCs, files). If the job needs deployment, TakTuk is used by Kadeploy as well and in the same manner.

TakTuk can be downloaded from its website [56], but it is also available in the official Debian [57] repositories.

5.2. **Managing the Configuration: Kameleon**

As a condition to obtain repeatable experiments and reproducible results, experimenters must manage their experiment's software environment, which includes the operating system, libraries,

runtimes, applications, and data. Infrastructures such as Grid'5000 or Emulab provide basic tools and procedures to help experimenters archive and deploy software environment. But experimenters often need deep control when building complex experiments. For instance, after running an experiment, the experimenter can sometimes detect a problem in the software environment that implies the need to rebuild a large part of it. Usually the build or rebuild of environment is done manually. Such an approach is not satisfactory, however. It is not only time consuming but also error-prone and can impair reproducibility. At a minimum, the experimenter must keep and correctly catalogue a trace of the environment build in order to be able to rebuild the environment in the same way. Another concern is versioning and access to the packages and data used to construction environment, to ensure its reconstruction and hence the experiment reproducibility. This issue is similar to the data provenance problem, a common issue in e-Science [58].

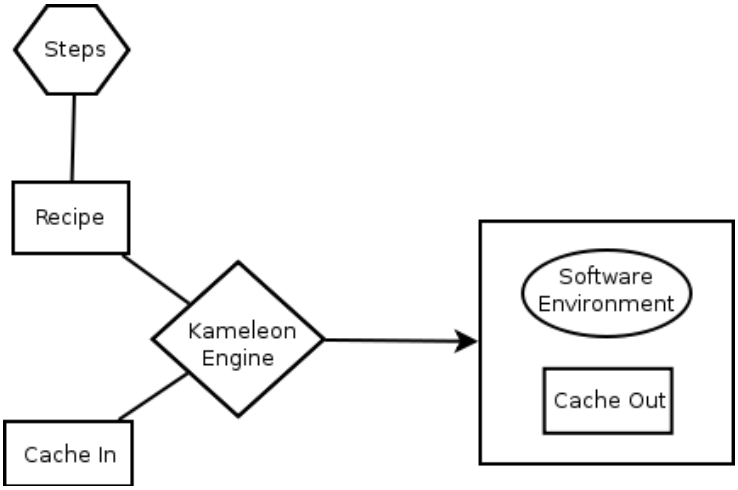

Figure 8: Environment generation with Kameleon

Kameleon is a tool developed to facilitate the building and rebuilding of the software environment. Figure 8 gives on overview of the environment generation process provided by Kameleon. The core engine loads a recipe file and retrieves the steps (mainly Unix commands) to execute. Next, the steps are processed in their order of appearance in the recipe. Kameleon then produces as output the environment generated and a cache archive containing data used to build the environment. This cache archive is used to keep all packages and data needed to rebuild the environment.

The closest tool to Kameleon is CDE [59], which generates an autonomous package with all binaries and data used during the execution of an application. This package allows the execution of an application on another machine without the need to worry about software dependencies. Its main difference from Kameleon is that the rebuilding issue is not considered.

5.3. **Reproducible Environment Creation: cloudinit.d**

An alternative method for repeatable experimental environment creation is represented by cloudinit.d [60], a tool for launching, configuring, monitoring, and repairing a set of interdependent virtual machines (VMs) in an infrastructure-as-a-service cloud or over a set of IaaS clouds from different cloud providers. A complex distributed experiment may have to be relaunched many times a day by several people, either repeating or refining the experiment. Cloudinit.d is designed to support the deployment and management of such complex distributed configurations.

The most essential feature of cloudinit.d is support for repeatable, one-click deployment of sets of VMs potentially including multiple services that might be involved in an experiment, such as storage, databases or identity servers. Those services can be deployed and redeployed easily and frequently, potentially by different actors (i.e., different researchers validating or evolving an experiment). In order to achieve consistent behavior, VM launches are based on a launch plan that can be created once, refined, and version controlled as the experiment evolves, and then executed many times to recreate the same conditions.

Part of the complexity involved in deploying experiments is that services within one launch can be interdependent so that information required for the deployment of one can be provided as a result of the deployment of another. For example, a service may need to know the hostname of a database server to complete its launch sequence: in this case the database server needs to be deployed first and the information about the hostname conveyed. On the other hand, services can also be independent: in this case the services can be deployed concurrently to save time. For this reason, cloudinit.d (much like the Unix init.d process) divides the service launch into run levels composed of independent services that accommodate both scenarios; each run level can define and resolve attributes to values that can be used by services launched in downstream run levels.

Many deployments move between different infrastructure or cloud providers, sometimes to create a testbed with specific properties, or example, a set of controlled widely distributed resources as in [21]. For this reason, cloudinit.d is platform-agnostic so that it can be deployed on any IaaS cloud or can integrate noncloud resources. For clouds, this is achieved via the use of adapters, in our case libcloud[61]. For noncloud providers, repeatability of environment build has to rely on the provider preserving the same base environment, that is, the operating system installation. This is typically feasible in testbeds that assume user control over those environments.

To deal with complex launches in a structured way and be able to reason about a complex system, cloudinit.d allows a user to make and verify assertions about vital properties of the system. Those assertions can be both generic (e.g., “Is the VM responding to pings”?) and user-defined (e.g., testing the setup of an application-specific property of a system). It is important that the user can define arbitrary soundness tests for the system. To this end, cloudinit.d allows users to select or configure such tests, associate them with services, and execute them to validate the correctness of a launch both at deployment time and running time. In order to ensure meeting a wide range of useful tests they are executed inside the environment (based on ssh into the environment) rather then rely on external information only.

In order to monitor the health of the experiment platform (or get feedback about potential irregularities), it is essential that the vital assertions about the launch can be reevaluated at any time. For this reason cloudinit.d allows users to rerun tests at any time by an action triggered automatically (e.g., during different stages of an experiment) or manually. The results of monitoring tests are stored in a database for experiment analysis and recreation. Further, if any of the assertions about the system (as embodied by the tests) fail, cloudinit.d allows the user to repair the launch components by applying a repair action defined by a policy. For example, a failure can lead to a number of repeats of a launch action or abandonment of a launch component or even the whole launch if a component is deemed to be irreparable.

6. Experiment Orchestration

FutureGrid supports several types of experiment orchestration atop a common set of basic services. The types can be broadly classified as interactive/scripted or batch/workflow.

In an interactive/scripted approach, users begin by running experiment-related commands interactively (typically by the command line) and explore the infrastructure and the experiments that they want to run. The next step is to execute entire experiments by using interactive

commands. Then, users can create scripts (e.g., shell scripts) to run entire experiments without any manual input. This approach is most suitable to users who expect to do a lot of interaction with their experiments and those that prefer writing scripts to specifying workflows.

In a batch/workflow approach, users begin by writing and executing trivial workflows to explore the infrastructure and their experiments. They then increase the complexity of their workflows until they are running workflows that are meaningful experiments. This approach is suitable to users who prefer writing workflows instead of scripts and users who expect to want to run a lot of unattended experiments.

The next subsections provide additional details about the tools that FutureGrid offers to support these two approaches.

6.1. **Interactive Experiment Management**

Interactive/scripted experiment management on a distributed testbed such as FutureGrid can be supported in a number of ways, but we focusing on two. The first approach is a Unix-style composition of command line tools. This approach is relatively simple; but similar to Unix as a whole, the composition of simple tools can be powerful.

Users perform five main tasks during interactive experiment management. The Unix-style approach of FutureGrid supports these tasks in the following ways:

- Users must provision resources so that they can access the resources needed for an experiment. To this end, they use command line tools to submit batch jobs (Torque qsub), reserve HPC resources (Moab mrsvctl), start virtual machines (Nimbus cloud-client.sh or cloudinit.d), and so on. The environment for the experiment is prepared by a combination of provisioning decisions (selecting the HPC systems and VM images that to start with) and executing distributed tasks (that complete node configuration).
- To perform an experiment, users must be able to execute distributed tasks. FutureGrid provides a new tool called the Host List Manager [62] to discover and add provisioned resources, organize those resources into groups, and generate a host list for each group. A parallel shell tool such as TakTuk [44] can then be used to execute commands on these lists of hosts.
- While an experiment is running, users need to monitor resources, services, and software for correctness. FutureGrid has deployed Inca [63] to monitor infrastructure and ensure that it is operating correctly. NetLogger [64] is also available to users who wish to instrument their software.
- In addition to correctness, users often want to observe performance during an experiment. The NetLogger tool is one way to do this. Vampir [65] is also available for fine-grained instrumentation and analysis. FutureGrid is in the process of deploying operating system kernels that support the PAPI [66] interface to hardware performance counters. In addition, FutureGrid has deployed Ganglia [67] on its resources so that users can easily obtain dynamic load information.
- After an experiment is complete, users want to store and share experimental results. One simple solution is to use the Unix script program to record sessions and save them to files. A simple way of sharing results is storing them in a shared filesystem.

FutureGrid is also developing a second approach to interactive experiment management based on messaging. This approach, called Message-based Execution and Monitoring System (MEMS), provides more functionality in a single tool and integrates a number of components into a messaging model. The integration makes it easy to store information about an experiment

(archive the message stream) as well as rerun an experiment (replay the command messages). A comparison of this approach with the Unix-style approach is shown in Table 4.

Table 4. Components of two approaches to interactive experiment management

Task	Unix-Style Tools	Messaging-Style Tools
Provision resources	Torque, Moab, Nimbus, Eucalyptus	Torque, Moab, Nimbus, Eucalyptus
Execute distributed tasks	Host List Manager, TakTuk	MEMS
Monitor for correctness	Inca, NetLogger	MEMS
Observe performance	NetLogger, Ganglia, Vampir, PAPI	MEMS, Vampir
Store and share results	Unix script, shared filesystem	Message archive

In the messaging approach, provisioning will still be done via the command line tools provided by the provisioning systems. The execution of distributed tasks will be accomplished by using the MEMS system. MEMS client programs will interact with the MEMS daemon running on each provisioned system using messages transmitted via a messaging service. Messages will be used to organize systems into groups, execute commands on groups of systems, and transmit the results of commands. Message-based experiments will be able to easily monitor correctness because FutureGrid will be publishing Inca results as messages and NetLogger already supports publishing information via messages.

Users can use this message-based approach to observe performance in many situations, but not all. The difficulty is that the messaging service can transmit several thousand messages a second, but detailed logging (particularly of parallel applications) can exceed this rate. In particular, Vampir logging of parallel applications can easily exceed this rate as can NetLogger. Users therefore will have to consider the data rate of their performance information when deciding whether they can transmit and store this information via MEMS.

Message-based experiments can easily store and share many of their results by using simple message archiving clients to store messages to a local file system. A slightly more complex approach is for users to request that an archival service store their messages.

6.2. **Using Pegasus for Experiment Management**

In the batched approach to conducting experiments, a workflow management system such as Pegasus enables the user to run multiple large-scale experiments simultaneously.

When applied to computer science, the *apparatus* is often a set of programs and execution environments based on the domain science. A virtual machine image is able to capture the most of the execution environment. The experiment itself often involves either processing massive parallel data in “proudly parallel” (what used to be called “embarrassingly parallel”) fashion or processing more complex interdependent steps in a workflow. The sensory data derives from log-files and other sources like monitoring. The Pegasus Workflow Management System is well suited to deal with these issues.

Pegasus [68] comprises a set of technologies that help workflow-based applications execute in a number of different environments including desktops, campus clusters, grids, and clouds. Pegasus enables scientists to construct workflows in abstract terms without worrying about the details of the underlying execution environment. The Pegasus Workflow Management Service maps scientific workflows onto available compute resources and executes the steps in their appropriate order. Pegasus can easily handle workflows from a single to several million computational tasks.

Pegasus has been used in a number of scientific domains including astronomy, bioinformatics, and geoscience. When errors occur, Pegasus tries to recover by using various strategies, from

automatically retrying single tasks, to providing a rescue workflow containing a description of the remaining work. Thus, after human intervention, the workflow can continue from the point of error.

Pegasus also keeps track of what has been done (provenance) including the locations of data used and produced and which software was used with which parameters. Provenance information is a necessary building block to capture the apparatus, by having sensors built into the experiment execution engine and thus facilitate repeatability. However, more work is necessary to completely describe an apparatus for experiment management.

Currently, Pegasus can use available resources, but it does not control them. The next step, therefore, is to prototype the provisioning and deprovisioning of resources into the workflow. A more formal workflow repository will hold experiments for other scientists to investigate and repeat. A rudimentary exchange of workflows is already available by virtue of Pegasus's workflow description being abstract.

7. Conclusions

The ability to design and support experiments is a vital but still little appreciated part of computer science. As we point out in Section 2, as the complexity and size of computer science systems grows, so does the need for experimentation to better understand and control them. Yet the experimental methodology for computer science so far has seen relatively little attention and, until recently, little investment.

The establishment of various experimental testbeds has created an opportunity to exchange lessons learned. For example, both Open Cirrus and Grid'5000 have developed meetings dedicated to sharing insights and discussion of experimental methodology as well as experimental results and have indicated the importance of such meetings in developing the community and experimental methods. FutureGrid, as the relative newcomer on the scene, may look to establish a similar mechanism.

One observation thus far is that the quality of research in computer science needs more attention, more structure, and more guidelines for computer science practitioners. Moreover, as noted in Section 4, testbeds represent the experimental feasibility and are thus closely linked to experimental methodology because we can only experiment in feasible ways. It is therefore logical that existing experimental testbeds should promote the development of experimental methodology in computer science.

Support for experimental science does not come cheaply: new custom capabilities and software must be developed, a higher than expected amount of effort must be devoted to supporting users with specialized requirements, and high levels of low utilization rates must be tolerated because of reconfiguration and availability needs. All these make support for experimental computer science more expensive than support for production science. At the same time, it is this level of support that allows computer scientists to focus on the experiment rather than searching to find enough hardware to validate their hypotheses at significant enough scales.

While some need for support is residual, some can be reduced by investment in better methods as well as sharing insights and technology within forums such as this workshop. Promising developments in the area are already taking place in tool reuse between FutureGrid and Grid'5000. For example, as noted in Sections 3.1 and 6.1, FutureGrid used the TakTuk software developed by Grid'5000 in its orchestration framework; and Grid'5000 used the Nimbus Infrastructure, sponsored by the FutureGrid project, to provide cloud computing for their experiments. Encouraging discussion and reuse where appropriate has positive results.

Cloud infrastructures play an increasingly larger role in all testbed infrastructures, both as an enabler and as a subject of study. The Open Cirrus testbed has been explicitly built to experiment with this new technology.

Overall, as discussed at the beginning of this report, computer science is an experimental science. In order to achieve the same activity and quality in experimental space that other experimental sciences achieve, it needs comparable investment. Furthermore, systematic development of experimental methodology is synergistic with the development of understanding on how to best organize computer science experiments and provide experimental infrastructure. The existing projects are well positioned to influence such development.

Acknowledgments

This material is based in part on work supported by the National Science Foundation under Grant No. 0910812 "FutureGrid: An Experimental, High-Performance Grid Test-bed", and in part by the Grid'5000 experimental testbed that has being developed under the INRIA ALADDIN development action with support from CNRS, RENATER, and several universities as well as other funding bodies (see https://www.grid5000.fr for details), and, in part, by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory (``Argonne") under Contract DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

1. *Workshop on Support for Experimental Computer Science*: http://graal.ens-lyon.fr/~desprez/SC11workshop.htm.
2. Denning, P.J., *Is Computer Science a Science?* Communications of the ACM, 2005. **48**(4): p. 27-31.
3. Denning, P.J., D.E. Comer, D. Gries, M.C. Mulder, A. Tucker, A.J. Turner, and P.R. Young, *Computing as a Discipline.* Communications of the ACM, 1989. **9-23**.
4. Denning, P.J., *ACM President's Letter: What is Experimental Computer Science?* Communications of the ACM, 1980. **23**(10): p. 543-544.
5. Paxson, V. and S. Floyd, *Wide-Area Traffic: The Failure of Poisson Modeling.* IEEE/ACM Transactions on Networking, 1995. **3**(3): p. 226-224.
6. Tichy, W., *Should Computer Scientists Experiment More?* IEEE Computer, 1998. **31**(5): p. 32-40.
7. Luckowicz, P., W. Tichy, E.A. Heinz, and L. Prechelet, *Experimental Evaluation in Computer Science: a Quantitative Study*. 1994, University of Karlshrue, Germany.
8. Zelkowitz, M.V. and D. Wallace, *Experimental Models for Validating Technology.* IEEE Computer, 1998. **31**(5): p. 23-31.
9. Team, A., *Algorithms for the Grid: INRIA Research proposal*: http://www.loria.fr/equipes/algorille/algorille2.pdf.
10. Jain, R.K., *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. 1991: Wiley.
11. Gustedt, J., E. Jeannot, and M. Quinson, *Experimental Methodologies for Large-Scale Systems: a Survey.* Parallel Processing Letter, 2009. **19**(3): p. 399-418.

12. Bolze, R., et al., *Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed.* International Journal of High Performance Computing Applications, 2006. **20**(4): p. 481-494.
13. Canon, L.-C., O. Dubuisson, J. Gustedt, and E. Jeannot, *Defining and Controlling the Heterogeneity of a Cluster: The Wrekavoc Tool.* Journal of System Software, 2010. **83**(5): p. 786-802.
14. Bailey, D., et al., *The NAS Parallel Benchmarks.* International Journal of High Performance Computing Applications, 1991. **5**(3): p. 63-73.
15. Casanova, H., A. Legrand, and M. Quinson. *SimGrid: a Generic Framework for Large-Scale Distributed Experiments*. in *10th IEEE International Conference on Computer Modeling and Simulation*. 2008.
16. *Grid'5000*: https://www.grid5000.fr.
17. Capello, F., et al. *Grid5000: a Large-scale, Reconfigurable, Controllable and Monitorable Grid Platform*. 2005: Grid'2005 Workshop.
18. Capit, N., G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. *A Batch Scheduler with High Level Componenets*. in *CCGrid 2005*. 2005.
19. Georgiou, Y., J. Leduc, B. Videau, J. Peyrard, and O. Richard. *A Tool for Environment Deployment in Clusters and light Grids*. in *2nd Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS)*. 2006.
20. Kleinjung, T., et al. *Factorization of a 768-bit RSA Modulus*. in *CRYPTO*. 2010. Berlin, Germany.
21. Riteau, P., M. Tsugawa, A. Matsunaga, J. Fortes, T. Freeman, D. LaBissoniere, and K. Keahey. *Sky Computing on FutureGrid and Grid'5000*. in *TeraGrid 2010*. 2010.
22. *Hemera*: https://www.grid5000.fr/Hemera.
23. *FutureGrid*: www.futuregrid.org.
24. *XSEDE: Extreme Science and Engineering Discovery Environment*: https://www.xsede.org/.
25. *OpenLDAP Web Site*.
26. *The Nimbus Toolkit*: www.nimbusproject.org.
27. *OpenStack: the open source, open standards cloud*: http://openstack.org/.
28. *The OpenNebula Project*: http://www.opennebula.org/.
29. *Eucalyptus*: http://eucalyptus.cs.ucsb.edu/.
30. *The Globus Project Web Site*.
31. *Uniform Interface to Computing Resource (UNICORE) Project*.
32. *Pegasus*: http://pegasus.isi.edu/.
33. von Laszewski, G., et al. *Design of the FutureGrid Experiment Management Framework*. in *GCE* 2010. New Orleans, LA.
34. Diaz, J.G., G. von Laszewski, F. Wang, A.J. Younge, and G. Fox. *FutureGrid Image Repository: A Generic Catalog and Storage System for Heterogeneous Virtual Machine Images*. in *Third IEEE International Conference on Coud Computing Technology and Science (CloudCom2011)*. 2010. Athens, Greece.
35. *FutureGrid Projects*: https://portal.futuregrid.org/projectssummary.
36. Campbell, R., et al. *Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research*. in *Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2009.
37. *Open Cirrus*: https://opencirrus.org/.
38. *Open Cirrus at Intel Labs Pittsburgh*: http://opencirrus.intel-research.net/.
39. Avetisyan, A.I., et al., *Open Cirrus: A Global Cloud Computing Testbed.* IEEE Computer, 2010(April, 2010): p. 35-43.
40. *TASHI*: http://incubator.apache.org/tashi/.
41. *Apache Hadoop*: http://hadoop.apache.org/.

42. *Maui Scheduler*: http://supercluster.org/maui.
43. Wang, G., D.G. Andersoen, M. Kaminsky, K. Papagiannaki, T.S.E. Ng, M. Kozuch, and M. Ryan. *c-Through: Part-time Optics in Data Centers*. in *Conference on Computer Communications (SIGCOMM 2010)*. 2010.
44. Claudel, B., G. Huard, and O. Richard. *TakTuk: Adaptive Deployment of Remote Executions*. in *HPDC*. 2009.
45. White, B., J. Leperau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Jaglekar. *An Integrated Experimental Environment for DIstirubted Systems and Networks*. in *OSDI*. 2002.
46. *Distem*: http://distem.gforge.inria.fr/)
47. Eide, E., L. Stoller, T. Stack, J. Freire, and J. Leperau. *Integrated Scientific Workflow Management for the Emulab Network Testbed*. in *ATEC*. 2006.
48. *The PlanetLab Project*: http://www.planet-lab.org/.
49. *Network eXperiment Engine (NXE): software to automate networking experiments in real testbeds*: http://www.ens-lyon.fr/LIP/RESO/Software/NXE/index.html.
50. *EXPO*: http://expo.gforge.inria.fr/index.html.
51. *Execo*: http://execo.gforge.inria.fr.
52. *G5K-Campaign*: http://g5k-campaign.gforge.inria.fr/.
53. Altintas, I., C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. *Kepler: An Extensible System for Design and Execution of Scientific Workflows*. in *16th Intl. Conference on Scientific and Statistical Database Management*. 2004.
54. *Taverna*: http://www.taverna.org.uk/.
55. *VisTrails*: http://www.vistrails.org/.
56. *TakTuk Website*: http://taktuk.gforge.inria.fr.
57. *Debian -- The Universal Operating System*: www.debian.org.
58. Simmhan, Y., B. Plale, and D. Gannon, *A Survey of Data Provenance in E-Science.* SIGMOD Record, 2005. **34**: p. 31-36.
59. Guo, P. *CDE: Run Any Linux Application On-Demand without Installation* in *USENIX Large Installation System Administration (LISA)*. 2011.
60. Bresnahan, J., T. Freeman, D. LaBissoniere, and K. Keahey, *Managing Appliance Launches in Infrastructure Clouds.* TeraGrid 2011, 2011.
61. *libcloud: a unified interface to the cloud*: http://incubator.apache.org/libcloud/.
62. *Interactive Experiment Management*: https://portal.futuregrid.org/tutorials/interactive-experiment-management.
63. Smallen, S., C. Olschanowsky, K. Ericson, P. Beckman, and J.M. Schopf. *The Inca Test Harness and Reporting Framework*. in *SC'2004 High Performance Computing, Networking, and Storage Conference*. 2004.
64. Gunter, D., B. Tierney, B. Crowley, M. Holding, and J. Lee. *NetLogger: A Toolkit for Distributed System Performance Analysis*. in *IEEE Mascots 2000: Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2000.
65. Nagel, W.E., A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources.* Supercomputer, 1996. **12**: p. 69-80.
66. Browne, S., J. Dongarra, N. Garner, and P. Mucci, *A Portable Programmign Interface for Performance Evaluation on Modern Processors.* The International Journal of High Performance Computing Applications, 2000. **14**(3): p. 189-204.
67. Federico, D., M.J.K. Sacerdoti, M.L. Massie, and D.E. Culler. *Wide Area Cluster Monitoring with Ganglia*. in *IEEE International Conference on Cluster Computing*. 2003: IEEE Press.

68. Deelman, E., J. Blythe, Y. Gil, C. Kesselman, G. Metha, S. Patil, M.-H. Su, K. Vahi, and M. Livny, *Pegasus: Mapping Scientific Workflows onto the Grid.* Proceedings of the 2nd Europena Across Grids Conference 2004.